Recursion and Backtracking Cheat Sheet

A concise guide to recursion and backtracking, fundamental algorithmic techniques, with examples and considerations for interview preparation.

Recursion Fundamentals

Basic Definition

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.

Essential components:

- **Base Case:** The condition that stops the recursion.
- **Recursive Step:** The function calls itself with a modified input.

Example: Factorial

Code:	Explanation:
`def factorial(n): if n == 0: return 1  # Base case else: return n * factorial(n-1) # Recursive step`	If `n` is 0, the function returns 1 (base case). Otherwise, it multiplies `n` by the factorial of `n-1`, recursively.

```python
def factorial(n):
  if n == 0:
    return 1  # Base case
  else:
    return n * factorial(n-1) # Recursive step
```

Call Stack

Each recursive call adds a new frame to the call stack. Deep recursion can lead to stack overflow errors if the base case is not reached or if the recursion is unbounded.

Understanding the call stack is crucial for debugging recursive functions. Visualize the call stack to trace the execution flow and identify potential issues.

Backtracking Techniques

Basic Definition

Backtracking is a problem-solving technique that incrementally builds candidates to the solutions, and abandons a candidate (“backtracks”) as soon as it determines that the candidate cannot possibly lead to a valid solution.

Core idea: Explore all possible solutions by trying every option. If a solution doesn’t work, revert to the previous state and try a different option.

Algorithm Steps

1. **Choose:** Select an option from the available choices.
2. **Explore:** Recursively explore the consequences of that choice.
3. **Unchoose:** If the choice doesn’t lead to a solution, undo the choice and try another.

Example: N-Queens

Problem Statement:	Place N chess queens on an N×N chessboard so that no two queens threaten each other.
Approach:	Try placing queens one by one in each row. If a placement leads to a conflict, backtrack and try a different column.
Key Ideas:	• Use recursion to explore possible placements. • Use helper functions to check if a placement is safe (no conflicts). • Backtrack by removing a queen if it leads to a dead end.

Recursion vs. Iteration

Comparison

Recursion:	Iteration:
• Elegant and concise for certain problems. • Can be less efficient due to function call overhead. • Easier to read for problems with a recursive structure.	• Generally more efficient in terms of performance. • Can be more complex to implement for recursive problems. • Avoids the risk of stack overflow.

When to use Recursion?

Use recursion when the problem has a natural recursive structure, such as tree traversal, graph algorithms, or problems that can be easily broken down into smaller, self-similar subproblems.

Consider iteration if performance is critical or if the recursion depth is likely to be large.

Tail Recursion

Tail recursion is a special form of recursion where the recursive call is the last operation in the function. Some compilers can optimize tail recursion into iterative code, avoiding stack overflow. However, Python does not optimize tail recursion.

Interview Strategies

Identifying Recursion/Backtracking Problems

Look for problems that involve searching, exploring combinations, or making choices at each step. Common keywords include “combinations”, “permutations”, “subsets”, “paths”, and “search”.

Structuring Your Solution

1. **Define the base case:** What condition stops the recursion?
2. **Define the recursive step:** How does the function call itself with a smaller subproblem?
3. **Handle edge cases:** Consider empty inputs or invalid states.

Optimization Techniques

Memoization:	Store the results of expensive function calls and reuse them when the same inputs occur again. Useful for overlapping subproblems (Dynamic programming).
Pruning:	Eliminate branches of the search space that cannot lead to a valid solution. Reduces the number of recursive calls.